COMP4075: Lecture 12 & 13

The Threepenny GUI Toolkit

Henrik Nilsson

University of Nottingham, UK

What is Threepenny (1)

- Threepenny is a GUI framework written in Haskell that uses the web browser as a display.
- A program written with Threepenny is a small web server that:
  - displays the UI as a web page
  - allows the HTML ***Document Object Model*** (DOM) to be manipulated
  - handles JavaScript events in Haskell
- Works by sending JavaScript code to the client.

What is Threepenny (2)

- Frequent communication between browser and server: Threepenny is best used running on localhost or over the local network.
- Written by Heinrich Apfelmus.

Rich API

- Full set of widgets (buttons, menus, etc.)
- Drag and Drop
- HTML elements
- Support for CSS
- Canvas for general drawing
- Functional Reactive Programming (FRP)

Conceptual Model

- Build and manipulate a Document Object Model (DOM): a tree-structured element hierarchy representing the document displayed by the browser.
- Set up event handlers to act on events from the elements.
- Knowing a bit of HTML helps.

The UI Monad

Most work take place in the the ***User Interface*** monad UI:

- Wrapper around IO; keeps track of e.g. window context.
- Instance of MonadIO, meaning that any IO operation can be lifted into UI:

$$liftIO :: IO\ a \to UI\ a$$

The Browser $Window$

- Type $Window$ represents a browser window.
- It has an attribute $title$ that may be written:

$$title :: WriteAttr\ Window\ String$$

- Retrieving the current window context:

$$askWindow :: UI\ Window$$

- Window passed to GUI code when server started:

$$startGUI :: Config \to (Window \to UI\ ()) \to IO\ ()$$

Elements

DOM made up of elements:

$$mkElement :: String \to UI\ Element$$

An element ***created*** when action run. Argument is an HTML elemen name: `"div"`, `"h1"`, `"p"`, etc.

Standard elements predefined:

$$\begin{aligned} div &:: UI\ Element \\ h1 &:: UI\ Element \\ br &:: UI\ Element \\ button &:: UI\ Element \end{aligned}$$

Attributes (1)

Elements and other entities like windows have attributes that can be read and written:

$$\begin{aligned} &\mathbf{type}\ Attr\ x\ a = ReadWriteAttr\ x\ a\ a \\ &\mathbf{type}\ WriteAttr\ x\ i = ReadWriteAttr\ x\ i\ () \\ &\mathbf{type}\ ReadAttr\ x\ o = ReadWriteAttr\ x\ ()\ o \end{aligned}$$

$$set :: ReadWriteAttr\ x\ i\ o \to i \to UI\ x \to UI\ x$$

$$get :: ReadWriteAttr\ x\ i\ o \to x \to UI\ o$$

$ReadWriteAttr$, $WriteAttr$ etc. are records of functions for attribute reading and/or writing.

set and get work for any type of entity.

Attributes (2)

Sample attributes:

$$title :: WriteAttr\ Window\ String$$
$$color :: WriteAttr\ Element\ String$$
$$children :: WriteAttr\ Element\ [Element]$$
$$value :: Attr\ Element\ String$$
$$(\#+) :: UI\ Element \to [UI\ Element] \to UI\ Element$$
$$(\#.) :: UI\ Element \to String \to UI\ Element$$

$(\#+)$ appends children to a DOM element.
$(\#.)$ sets the CSS class.

Attributes (3)

Example usage ($(\#)$ is reverse function application):

```
mkElement "div"
  # set style     [("color", "#CCAABB")]
  # set draggable True
  # set children otherElements
```


Events (1)

- The type $Event\ a$ represents streams of time-stamped events carrying values of type a.
- Semantically: $Event\ a \approx [(Time, a)]$
- $Event$ is an instance of $Functor$.
- $Event$ is ***not*** an instance of $Applicative$. The type for $<\!*\!>$ would be

  $$Event\ (a \to b) \to Event\ a \to Event\ b$$

  However, this makes no sense as event streams in general are not synchronised.

Events (2)

Most events originate from UI elements; e.g.:

- $valueChange :: Element \to Event\ String$
- $click :: Element \to Event\ ()$
- $mousemove :: Element \to Event\ (Int, Int)$ (coordinates relative to the element)
- $hover :: Element \to Event\ ()$
- $focus :: Element \to Event\ ()$
- $keypress :: Element \to Event\ Char$

Events (3)

One or more handlers can be registered for events:

$$register :: Event\ a \to Handler\ a \to IO\ (IO\ ())$$

The resulting action is intended for deregistering a handler; future functionality.

Events (4)

Usually, registration is done using convenience functions designed for use directly with elements and in the UI monad:

$$on :: (element \to Event\ a)$$
$$\to element \to (a \to UI\ void) \to UI\ ()$$

For example:

```
do
  ...
  on click element $ λ_ → ...
  ...
```


Behaviors (1)

- The type $Behavior\ a$ represents continuously time-varying values of type a.
- Semantically: $Behavior\ a \approx Time \to a$
- $Behavior$ is an instance of $Functor$ ***and*** $Applicative$.
- Recall that events are not an applicative. However, the following provides similar functionality:

  $$(<\!@\!>) :: Behavior\ (a \to b)$$
  $$\to Event\ a \to Event\ b$$

Behaviors (2)

- Attributes can be set to time-varying values:

  $$sink :: ReadWriteAttr\ x\ i\ o$$
  $$\to Behavior\ i \to UI\ x \to UI\ x$$

- There is also:

  $$onChanges :: Behavior\ a$$
  $$\to (a \to UI\ void) \to UI\ ()$$

  But conceptually questionable as a behavior in general is ***always*** changing.

FRP (1)

Threepenny offers support for Functional Reactive Programming (FRP): transforming and composing behaviours and events as “whole values”.

For example:

- $filterJust :: Event\ (Maybe\ a) \to Event\ a$
- $unionWith :: (a \to a \to a)$ $\to Event\ a \to Event\ a \to Event\ a$
- $unions :: [Event\ a] \to Event\ [a]$
- $split :: Event\ (Either\ a\ b) \to (Event\ a, Event\ b)$

FRP (2)

- `accumE :: MonadIO m => a -> Event (a -> a) -> m (Event a)`
- `accumB :: MonadIO m => a -> Event (a -> a) -> m (Behavior a)`
- `stepper :: MonadIO m => a -> Event a -> m (Behavior a)`
- `(<@>) :: Behavior (a -> b) -> Event a -> Event b`

Note: Stateful events and behaviors are returned as monadic computations.

Hello World (1)

A simple "Hello World" example:

- Display a button
- Change its text when clicked

First import the module. Large API, so partly qualified import recommended:

```haskell
module Main where

import qualified Graphics.UI.Threepenny as UI
import           Graphics.UI.Threepenny.Core
```


Hello World (2)

The *startGUI* function starts a server:

```haskell
startGUI :: Config -> (Window -> UI ()) -> IO ()
```

- *Config*-records carry configuartion parameters.
- *Window* represents a browser window.
- The function `Window -> UI ()` is called whenever a browser connects to the server and builds the initial HTML page.

Hello World (3)

Start a server listening on port 8023; static content served from `../wwwroot`:

```haskell
main :: IO ()
main = do
  startGUI
    defaultConfig
      { jsPort   = Just 8023,
        jsStatic = Just "../wwwroot"}
    setup
```


Hello World (4)

Start by setting the window title:

```haskell
setup :: Window -> UI ()
setup window = do
  return window # set UI.title "Hello World!"
```

Reversed function application: `(#) :: a -> (a -> b) -> b`

set has type:

```haskell
set :: ReadWriteAttr x i o -> i -> UI x -> UI x
```

The window reference is a pure value, passed in, hence the need to lift it into a *UI* computation using *return*.

Hello World (5)

Then create a button element:

```haskell
button <- UI.button # set UI.text "Click me!"
```

Note that *UI.button* has type:

```haskell
UI.button :: UI Element
```

A new button is ***created*** whenever that action is run.

DOM elements can be accessed much like in JavaScript: searched, updated, moved, inspected.

Hello World (6)

To display the button, it must be attached to the DOM:

```haskell
getBody window #+ [element button]
```

The combinator `(#+)` appends DOM elements as children to a given element:

```haskell
(#+) :: UI Element -> [UI Element]
       -> UI Element
```

getBody gets the body DOM element:

```haskell
getBody :: Window -> UI Element
```

Here, *element* is just *return*.

Hello World (7)

Finally, register an event handler for the click event to change the text of the button:

```haskell
on UI.click button $ const $ do
  element button
    # set UI.text "I have been clicked!"
```

Types:

```haskell
on :: (element -> Event a) -> element
     -> (a -> UI void) -> UI ()
UI.click :: Element -> Event ()
```


Buttons (1)

```haskell
mkButton :: String -> UI (Element, Element)
mkButton title = do
  button <- UI.button #. "button" #+ [string title]
  view <- UI.p #+ [element button]
  return (button, view)

mkButtons :: UI [Element]
mkButtons = do
  list <- UI.ul #. "buttons-list"
  ...
```

Buttons (2)

```
(button1, view1) ← mkButton button1Title
on UI.hover button1 $ \_ → do
  element button1 # set text (button1Title ++ " [hover]")
on UI.leave button1 $ \_ → do
  element button1 # set text button1Title
on UI.click button1 $ \_ → do
  element button1 # set text (button1Title ++ " [pressed]")
  liftIO $ threadDelay $ 1000 ∗ 1000 ∗ 1
  element list
    #+ [UI.li # set html "<b>Delayed</b> result!"]
```


Buttons (3)

```
(button2, view2) ← mkButton button2Title
on UI.hover button2 $ \_ → do
  element button2 # set text (button2Title ++ " [hover]")
on UI.leave button2 $ \_ → do
  element button2 # set text button2Title
on UI.click button2 $ \_ → do
  element button2 # set text (button2Title ++ " [pressed]")
  element list
    #+ [UI.li # set html "Zap! Quick result!"]
return [list, view1, view2]
```


Counter Example 1 (1)

Simple counter, basic imperative style.

Idea:

- Keep the count in an imperative variable
- The click event handler increments the counter and updates the display accordingly.

Counter Example 1 (2)

```
setup :: Window → UI ()
setup window = do
  return window
    # set UI.title "Counter Example 1"
  let initCount = 0
  counter ← liftIO $ newIORef initCount
  button  ← UI.button # set UI.text "+1"
  label   ← UI.label # set UI.text
                ("Count: " ++
                 show initCount)
```


Counter Example 1 (3)

```
getBody window #+ [UI.center
                      #+ [element button,
                          UI.br,
                          element label]]
on UI.click button $ const $ do
  count ← liftIO $ do
    modifyIORef counter (+1)
    readIORef counter
  element label # set UI.text ("Count: " ++
                                show count)
```


Counter Example 2 (1)

Counter with reset, “object-oriented” style.

Idea:

- Make a counter object with encapsulated state and two operations: reset and increment.
- Make a display object with a method for displaying a value.

Counter Example 2 (2)

Make a counter object:

```
mkCounter :: Int → UI (UI Int, UI Int)
mkCounter initCount = do
  counter ← liftIO $ newIORef initCount
  let reset = liftIO $ writeIORef counter initCount
                ≫ return initCount
      incr  = liftIO $ modifyIORef counter (+1)
                ≫ readIORef counter
  return (reset, incr)
```


Counter Example 2 (3)

Make a display object:

```
mkDisplay :: Int → UI (Element, Int → UI ())
mkDisplay initCount = do
  let showCount count =
        "Count: " ++ show count
  display ← UI.label # set UI.text
                (showCount initCount)
  let dispCount count =
        () <$ element display
                # set UI.text (showCount count)
  return (display, dispCount)
```


Counter Example 2 (4)

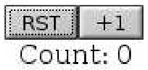
```
setup :: Window → UI ()
setup window = do
  return window
    # set UI.title "Counter Example 2"
  let initCount = 0
  (reset, incr) ← mkCounter initCount
  (display, dispCount) ← mkDisplay initCount
  buttonRst ← UI.button # set UI.text "RST"
  buttonInc ← UI.button # set UI.text "+1"
```

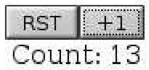
Counter Example 2 (5)

```
getBody window
  #+ [UI.center #+ [element buttonRst,
                    element buttonInc,
                    UI.br,
                    element display]]
on UI.click buttonRst $ const $ reset >>= dispCount
on UI.click buttonInc $ const $ incr >>= dispCount
```


Counter Example 3 (1)

Counter with reset, FRP style.





Idea:

- Accumulate the button clicks into a ***time-varying*** count; i.e., a *Behavior Int*.
- Make the text attribute of the display a time-varying text directly derived from the count; i.e., a *Behavior String*.

Counter Example 3 (2)

```
setup :: Window → UI ()
setup window = do
  return window
    # set UI.title "Counter Example 3"
  let initCount = 0
  buttonRst ← UI.button # set UI.text "RST"
  buttonInc ← UI.button # set UI.text "+1"
  let reset  = (const 0)  <$ UI.click buttonRst
  let incr   = (+1)       <$ UI.click buttonInc
```

Note: *Event* and *Behavior* are instances of *Functor*.

Counter Example 3 (3)

```
count   ← accumB 0 $ unionWith const reset incr
display ← UI.label
            # sink UI.text
                (fmap showCount count)
```

Type signatures:

```
accumB :: MonadIO m ⇒
             a → Event (a → a) → m (Behavior a)
unionWith :: (a → a → a)
               → Event a → Event a → Event a
sink :: ReadWriteAttr x i o
          → Behavior i → UI x → UI x
```


Counter Example 3 (4)

```
getBody window
  #+ [UI.center #+ [element buttonRst,
                    element buttonInc,
                    UI.br,
                    element display]]
```

- No callbacks.
- Thus no "callback soup" or "callback hell"!
- Fairly declarative description of system: ***Whole-value Programming***.
- This style of programming has had significant impact on programming practice well beyond FP.

Currency Converter (1)

```
return window # set title "Currency Converter"
dollar ← UI.input
euro ← UI.input
getBody window #+ [
  column [
    grid [[string "Dollar:", element dollar]
         ,[string "Euro:", element euro]]
  , string "Amounts update while typing."
  ]]
```


Currency Converter (2)

```
euroIn ← stepper "0" $ UI.valueChange euro
dollarIn ← stepper "0" $ UI.valueChange dollar
let
  rate = 0.7 :: Double
  withString f =
      maybe "-" (printf "%.2f") ∘ fmap f ∘ readMay
  dollarOut = withString (/rate)  <$>  euroIn
  euroOut = withString (*rate)  <$>  dollarIn
element euro # sink value euroOut
element dollar # sink value dollarOut
```


Reading

- Overview, including references to tutorials and examples:
  `http://wiki.haskell.org/Threepenny-gui`
- API reference:
  `http://hackage.haskell.org/package/threepenny-gui`